涉密论文 □　公开论文 ☑

浙 江 大 學

本科生毕业论文

题目　知识驱动的物联网嵌入式固件自动化根因分析方法研究

姓名与学号　张乔 3200102817

指导教师　纪守领

年级与专业　2020级 计算机科学与技术

所在学院　计算机科学与技术学院

递交日期　递交日期

浙江大学本科生毕业论文（设计）承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2. 本人在毕业论文（设计）中除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得**浙江大学**或其他教育机构的学位或证书而使用过的材料。

3. 与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

4. 本人承诺在毕业论文（设计）工作过程中没有伪造数据等行为。

5. 若在本毕业论文（设计）中有侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

6. 本人完全了解**浙江大学**有权保留并向有关部门或机构送交本论文（设计）的复印件和磁盘，允许本论文（设计）被查阅和借阅。本人授权**浙江大学**可以将本论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编本论文（设计）。

作者签名：　　导师签名：

签字日期：　　年　月　日　　签字日期　　年　月　日

致 谢

摘 要

Abstract

目录

第一部分　毕业论文

第二部分　毕业论文开题报告

第一部分

毕业论文

1 绪论

1.1 课题背景与意义

随着硬件技术和网络通信技术的快速发展，越来越多的设备和系统被连接到互联网，实现了智能化的生活和工作环境。嵌入式固件是物联网设备的核心，它使得这些设备能够实现数据采集、通信、控制等功能，从而为用户提供更便捷、智能的服务体验。同时物联网技术的应用已经渗透到各个产业领域，包括国防、工业、家居、医疗、交通等行业。嵌入式固件在这些领域中起着关键作用，它使得设备和系统能够实现自动化、智能化的控制和管理，提高生产效率、节约资源、降低成本。随着物联网设备的普及，信息安全和隐私保护越来越受到学术界与工业界的关注。嵌入式固件作为物联网设备的核心软件，需要具备良好的安全性和防护能力，以防止未经授权的访问、数据泄露、设备篡改等安全威胁，因此需要对其安全性进行检查。IBM Security 发布的《2022 年数据泄露成本报告》[1]表明，关键性基础设施在采用零信任安全策略（Zero Trust）方面还很滞后，其数据泄露的平均成本高达 540 万美元，比已采用零信任策略的组织高出 117 万美元。而关键基础设施中含有着众多的固件，这无疑说明固件安全是继续考虑的。HP Wolf security 发布 Threat Insights Report Q1 2022[2]表明，的随着劳动力的去中心化和混合办公模式的流行，端点安全的管理模式被彻底打破。设备不在现场供 IT 团队访问的混合办公环境中。更多的端点位于企业网络的保护之外也会降低可见性并增加通过不安全网络访问企业内部网络的攻击风险。购买远程办公设备的办公室工作人员中有 68% 表示安全不是他们采购决策的主要考虑因素。此外，43% 的人没有让 IT 或安全部门检查或安装他们的新笔记本电脑或 PC。如果不关注固件安全，这些多端的固件存在着严重的风险。

模糊测试技术可以通过将自动或半自动生成的测试输入到程序中，监测程序是否发生崩溃；而根因分析技术可以根据模糊测试的结果得到导致程序崩溃的测试输入，并寻找到程序中存在漏洞的位置。然而，目前尚未有针对运行资源受限的物联网嵌入式固件的自动化根因分析方法，无法针对嵌入式固件崩溃进

行深入分析。同时为了满足低级硬件中受约束的计算资源的需求，嵌入式固件通常被剥离调试信息，调试信息通常包含了变量名、函数名、文件名、行号等关键信息，有助于理解代码的结构和逻辑。当这些信息被剥离时，只留下一堆混合着数据的连续指令，这使得分析人员将失去对代码执行上下文的理解，更难理解代码的语义，从而导致手动根因分析变得更加困难。同时由于逆向执行时存在内存别名等问题，同时难以得到程序运行时的控制流与数据流，传统的自动化根因分析工具难以生效。

综上所述，针对物联网嵌入式固件设备，尚未存在有效的自动化根因分析方法衔接模糊测试，无法确定固件发生崩溃崩溃的根本原因并加以修正。因此，设计针对嵌入式固件的自动化根因分析工具是当前急需解决的问题。

1.2 国内外研究现状

1.2.1 嵌入式固件模糊测试技术

嵌入式固件运行在各种硬件平台上[3][4]，并与多个复杂的外围设备交互。早期的一些研究直接在真实硬件设备上进行黑盒 fuzzing，但这种方法缺乏全面的覆盖引导。例如，μAFL[5]和 SyzTrust[6]利用硬件调试器收集运行时信息，并支持嵌入式跟踪宏单元（ETM[7]）功能。其他研究采用硬件在环（hardware-in-the-loop）方法[8]将硬件请求转发到真实的外围设备。然而，这些方法需要频繁的上下文切换和模拟器与硬件之间的状态同步，导致显著的性能开销。

近期的工作采用重宿主技术以提高可扩展性，这种技术的核心思想是模拟外设并为固件提供有效输入，主要帮助固件通过初始化阶段的状态检查这些重宿主工作主要专注于使 IoT 固件能够进行 fuzzing，而不是专门优化后端 fuzzers（例如 AFL[9]）以增强 fuzzing 过程的效果。

1.2.2 自动化故障定位技术

虽然模糊测试有助于发现大量导致崩溃的测试用例，揭示潜在的漏洞，但识别这些崩溃的根本原因所需的最终手动调查是一个耗时且乏味的过程。这促使

了自动化故障定位技术的发展。

频谱基方法通过分析崩溃和非崩溃测试用例，根据它们出现的频率来对可疑指令进行排名[10][11][12]。虽然有效，但这些方法在分析粒度上存在限制。更先进的技术探索初始崩溃附近的不同执行路径，并为程序实体分配分数[13][14]。然而，分析每个个别测试用例的重时间成本使它们对于大规模固件 fuzzing 来说不切实际。

基于事后的方法利用崩溃后的遗留物，如执行轨迹和内存转储，反向分析程序并识别可能负责崩溃的最小指令集。例如，CrashLocator[15]利用崩溃报告中的崩溃栈信息定位故障函数，RETracer[16]反向分析代码和栈以找出错误值的传播方式。

此外，基于学习的方法也被广泛研究，以更好地理解崩溃。这些方法提出了神经网络架构来分析并将程序上下文纳入可疑分数计算。最近，随着大型语言模型（LLMs）的兴起，大模型辅助进行根因分析领域取得了积极的进展[17][18]

1.3 研究的主要难点

运行时资源有限 嵌入式固件通常部署在资源受限的设备上，这些设备缺乏足够的调试机制。理想情况下，根本原因分析方法需要对崩溃时的运行时信息有全面的了解，如执行轨迹、寄存器和内存值等。以往的研究通常依赖于核心转储来恢复这些运行时信息，但这严重依赖于崩溃时未损坏的栈数据和部分内存。与操作系统相比，嵌入式固件缺少用于检测崩溃（例如，sanitizers）和记录运行时信息（例如，核心转储）的机制。有时，开发者甚至需要采用原始的方法手动监控设备状态和通信，如 LED 闪烁和串行打印[19]。

调试信息有限 嵌入式固件通常被简化为原始二进制文件，以满足低级硬件中受限的计算资源需求。在此过程中，调试符号（如函数和变量名称）会被剥离，只留下一连串的指令和数据，这使得分析人员难以理解代码的语义。剥离后的固件二进制文件中缺乏语义信息，使得分析人员面对大量可能可疑的指令，而这些指令仍需进行繁琐的手动调查。然而，以往的工作要么将所有指令一视同仁，

要么设计上无法区分同一基本块内的不同指令，这对最终的手动根本原因调查提供了有限的实际指导。

1.4 研究内容与贡献

本文研究的主要内容是物联网嵌入式固件自动化根因分析方法。本文提出了一种针对物联网固件程序的框架，针对上节所提到的难点，本文进行了设计研究，主要贡献如下：

（1）设计了一种高效的运行时记录方法。针对嵌入式固件运行时资源有限以及调试信息有限的特点，我们设计了一种高效的运行时信息记录的方法。这种方法的核心思想是崩溃复现过程中，关注具体内存访问行为，并在故障定位中显著加速解决内存别名问题。具体内存访问包括指令的地址、访问方法（读或写）、源（目的地）寄存器，以及具体的目的地（源）内存地址。

（2）实现了 arm 平台下的自动化根因分析技术。我们设计了基于模糊测试结果的针对 arm 平台下的物联网嵌入式固件的根因分析方法。通过设计了一种高效运行时记录方法，构造使用-定义链，以及逆向执行等方法，分析模糊测试中的崩溃案例，分析导致崩溃的根本原因，定位固件漏洞

1.5 本文的组织结构

本文针对物联网嵌入式固件自动化根因分析方法这一问题，对现有自动化根因方法进行总结分析，提出了一种针对嵌入式固件的自动化分析方法，实现了对模糊测试中发生崩溃的固件漏洞检测。

本文共分为 5 章，其中各个章节具体内容安排如下：

第一章为绪论，介绍本文所研究的物联网嵌入式固件自动化根因分析方法这一主题的研究背景、意义与研究现状，并指出了研究难点与解决思路。

第二章为相关技术研究，介绍了主流自动化根因分析方法与物联网嵌入式固件自动化分析方法所面临的问题与解决方法。

第三章为本文主要工作，首先对物联网嵌入式固件自动根因分析方法框架

进行概述，然后并进行了问题定义与别名问题分析。最后从高效的运行时信息记录方法，使用定义链与逆向执行三个主要设计结构等三个方面对本文设计的方法进行介绍。

第四章为实验与评估，介绍本文所提出的方法所使用的实验环境，给出对于物联网嵌入式固件自动根因分析方法。

第五章为总结与展望，总结本文主要工作与贡献，提出本文工作中的局限性，并思考后续在此基础上可以继续发展与设计的工作

1.6 本章小结

本章中，我们首先指出了物联网嵌入式固件的重要性，分析了其自身存在的安全问题，论证了自动化分析方法研究的重要性。之后我们介绍了目前对于物联网嵌入式固件自动化根因分析方法研究的现状以及所面临的主要难点，讨论了当前主要研究方法的不足，思考了解决思路与后续可能的研究趋势，并介绍了本文的研究内容与贡献。最后我们介绍了本文的章节安排与组织结构。

2 自动化根因分析相关技术研究

2.1 概述

在科学和工程学中，根因分析是一种解决问题的方法，用于识别故障或问题的根本原因。它广泛应用于 IT 运营、电信、工业过程控制、事故分析、医学等。根因分析通常是一种系统性的过程，需要收集和分析与问题相关的数据和信息，运用各种工具和技术来确定问题的根本原因。在计算机领域，特别是在软件安全领域，根因分析通常是指分析软件或系统崩溃的根本原因，它可能涉及对系统的代码、设计、配置、环境等方面进行深入分析。通过根因分析，开发团队可以更好地了解程序中的潜在问题，并及时采取措施修复和加固系统，以提高软件的安全性和稳定性。作为一种粗略的可靠性度量方法，模糊测试可以针对待分析程序产生不同输入得到程序崩溃，并提示程序哪些部件需要特殊的注意；

而根因分析作为模糊模糊测试的后续阶段，可以利用，模糊测试得到的已知崩溃进行深入分析并找出导致这些问题的根本原因。一般来说，当前自动化根因分析方法主要包括两类: 基于频谱的根因分析方法和基于事后的根因分析方法。其中基于频谱的方法通常在初始崩溃测试用例的基础上生成两大组崩溃测试用例和非崩溃测试用例。它们记录每条指令上的寄存器和内存数据，以统计方式构建崩溃的必要条件作为谓词，测量每个谓词与初始崩溃的相关性，最后将谓词列为根本原因；而基于事后的故障定位方法从调试文件 (例如核心转储和内存快照) 开始，执行反向执行和向后污染分析，以跟踪无效数据的传播。它们分析指令的语义，比基于频谱的故障定位方法更有效。同时，随着当前 AI 领域的发展，机器学习的长足发展与大模型的出现使根因分析进入了新的维度，一些繁杂的人工工作可以交由 AI 辅助进行。这极大提高了一些受到时间和复杂程度限制方法的效率，拓宽了根因分析领域

下面将从这三个方面介绍自动化根因分析相关技术的研究

2.2 自动化根因分析相关方法

2.2.1 基于频谱的自动化根因分析方法

基于频谱的分析方法是一种利用软件系统的执行频谱（Execution Spectrum）来识别和定位问题的技术。该方法通常用于识别性能问题和资源利用不足等情况。基本原理是在软件运行时记录系统的执行轨迹或事件，然后通过分析这些事件的频率和模式来识别可能的瓶颈或问题。基于频谱的故障定位技术基于一个核心假设：程序中出错频率较高的部分更有可能是故障所在。这种方法使用“频谱”信息，即程序执行过程中各个组件（如函数、语句或分支）的执行频率和失败频率的统计数据。例如，对于性能问题，可以记录系统中的函数调用、代码路径执行次数、资源利用情况等信息。通过分析这些数据，可以确定哪些函数或路径是性能瓶颈，并识别可能的优化方向。这种方法可以帮助开发人员了解系统的执行特征，并优化系统的性能和资源利用率。大多数基于频谱的根因分析技术研究侧重于统计量的设计，包括排名指标和分布统计。但也有一些工作侧

重对测试用例的研究：Hao 等人[20]根据测试用例的能力提出了三种减少测试用例数量的策略,基于使用测试输入的测试运行的执行跟踪，使开发人员可以只选择测试输入的一个有代表性的子集来进行结果检查和故障定位；Abreu 等人[21]通过使用由西门子集和空间程序组成的通用基准，研究了作为多个参数（例如系统执行期间收集的程序频谱的质量和数量）的函数的诊断准确性，结果表明用于分析程序谱的特定相似系数的优越性能在很大程度上独立于测试设计，并且证明了 SFL 可以有效地应用于工业环境中的嵌入式软件开发环境中；Dandan Xu 等人[22]设计并实现了一种反例强化学习技术，该技术奖励涉及反例的操作，通过平衡随机抽样和对反例的利用，利用每个模糊测试回合的结果来指导下一轮模糊测试，从而将当今基于频谱的根因分析工具的可扩展性和准确性提高了一个数量级以上。

2.2.2 基于事后的故障定位方法

基于事后的根因分析技术是一种在崩溃发生后开始的根因分析技术。该方法的基本原理是通过收集和分析系统状态、日志、内存转储等信息，来确定问题的根本原因和触发条件。通过分析这些信息，可以确定程序的内存泄漏、空指针引用等异常行为，并采取措施修复或预防类似问题的再次发生。在事后程序分析技术中，记录与重放[23][24][25]和核心转储分析[16][26][27]是两种常见方法。

记录与重放 这种方法的核心思想是在程序运行时使用工具捕获程序的执行轨迹和状态，从而允许开发者在出现故障时能够准确重现问题。在程序执行过程中，记录所有导致状态变化的事件，包括函数调用、外部输入、线程操作和其他系统调用等事件。之后利用捕获的这些事件的参数和结果，以及它们的时间戳和执行顺序进行事后分析，找到导致崩溃的根因。值得注意的是通常“记录”过程需要通过修改操作系统的内核、使用特定的库或工具、或者插桩代码来实现；同时“重放”过程通常需要在一个控制的环境中进行，以避免非确定性行为的干扰。这种方法存在着局限性：（1）记录阶段可能会引入显著的时间和空间开销，特别是在需要详尽记录的情况下。（2）无法重放某些不确定性较高的行为：

如并发和竞争条件漏洞导致的崩溃就可能在重放时难以精确复现。

核心转储分析 与记录与重放方法不同，核心转储分析方法涉及捕捉程序崩溃时的内存镜像，以便分析和诊断导致崩溃的原因。这种技术对于理解复杂软件系统的失败原因非常有用，特别是在调试难以复现的故障时体现出了较大的优势。核心转储方法的基本原理是当程序异常终止（如段错误）发生崩溃时，自动保存那部分程序执行时的内存内容。这个转储文件包含了程序终止时的变量值、程序计数器、寄存器内容、堆栈信息等关键数据。发生崩溃后，分析者通过检查崩溃时的内存状态，分析崩溃的上下文，确定崩溃发生在程序的哪一部分。这种方法存在着一定的局限性：（1）核心转储文件可能非常大，尤其是密集应用内存的程序，这可能极大提高分析的开销。（2）某些崩溃依赖于程序运行的特定环境或状态，单纯利用核心转储文件无法分析导致崩溃的原因。

虽然当前这些研究被证明是有效的，但它们只关注特定的类型状态问题，或者在反向执行时难以解决内存别名等问题，仍然不足够高效。

2.2.3 基于机器学习的自动化根因分析方法

这种方法利用机器学习算法来自动识别和诊断系统故障的根本原因，并依赖于历史数据和模式识别技术来预测和识别问题的根源。这种方法在处理复杂系统和大规模数据时显示出其独特的优势。通过分析系统的正常运行数据和故障时的数据，机器学习模型可以学习到哪些指标或事件与系统故障强相关。在基于机器学习的自动化根因分析方法中，首先收集系统的运行日志、性能指标、系统事件等数据；其次从原始数据中提取有用的特征如 CPU 使用率、内存使用量、响应时间、错误率等，这些特征能够代表系统的运行状态或可能影响系统性能和稳定性的因素；之后利用其训练机器学习模型，并在实际数据上运行模型，以检测和预测潜在的系统故障。这种方法也存在着一定的局限性：（1）模型的效果很大程度上依赖于高质量和高覆盖率的数据，需要人工提供较为优质的训练数据。（2）模型需要不断维护、微调与更新，从而能在快速变化的系统环境中适应新的数据和条件。图 n 展示了 ModelCoder[28]这一基于机器学习的自动化根因

分析方法的实现过程：

2.3 其他相关研究

2.3.1 逆向执行

在基于事后分析的故障定位领域，分析人员通过反向执行来更深入地了解程序崩溃[16][29]。反向执行是一种通过撤销已执行的指令,恢复寄存器和内存值到先前状态的技术，使分析人员能够从独特的视角观察程序状态随时间的演变过程。这需要分析人员设计复杂的处理器，根据指令的行为逆向执行指令。在本文关注的嵌入式固件领域，目标硬件平台的资源受限性质通常严重限制了监控和记录能力，导致无法获取足够的数据以支持全面的反向执行。为了解决这一问题，需要设计合理的运行时记录方法，并为嵌入式固件设计合适的逆向处理器。

2.3.2 使用——定义链

在数据流分析领域，使用——定义链 (Use-Define Chain) 是一种描述程序中一个变量的使用和所有定义的数据结构，该变量可以在没有任何其他中间定义的情况下到达使用[30]。使用定义链建立了程序中变量的定义点（定义变量值的地方）与使用点（使用这些变量值的地方）之间的关系。通过这种结构，分析人员可以追踪变量值是如何在程序中传递和转换的，从而可以提高逆向执行中的效率

2.4 本章小结

本章中，首先对根因分析方法进行了简单背景介绍，然后讨论了当前主流的几种自动化根因分析方法，并分析了他们的优势与不足。最后补充分析了自动化根因分析方法中逆向执行和使用——定义链等其他相关研究。

3 物联网嵌入式固件自动化根因分析

3.1 引言

随着网络技术的飞速发展，物联网技术的应用已经渗透到各个产业领域。嵌入式固件在这些领域中起着关键作用，它使得设备和系统能够实现自动化、智能化的控制和管理，提高生产效率、节约资源、降低成本。然而随着嵌入式固件深入生活各个领域，其中的漏洞与安全问题重要性愈发重要。2023 年 12 月，丰田雷克萨斯[31]由于发动机电子控制单元存在安全隐患，召回了超过十万辆车。嵌入式固件安全在国防方面同样重要，俄乌战争中，Viasat 的 KA-SAT 网络收到了攻击[32]，攻击者利用卫星调制解调器固件的漏洞进行攻击，导致 KA-SAT 的卫星宽带服务部分中断，影响了数千名客户和 5800 台风力涡轮机的远程访问。

目前模糊测试领域已有较为成熟的方法来检测固件漏洞，但在模糊测试发生崩溃后，需要进一步进行根因分析导致崩溃的原因，修复漏洞。目前已有几种主流的自动化根因分析方法。但目前尚未有针对嵌入式固件的自动化根因分析方法。由于嵌入式固件运行时资源受限，以及代码缺乏必要的调试信息，导致在固件领域已有的自动化分根因析方法难以有效分析导致崩溃的根本原因；而手动分析较为繁琐，且由于人工难以理解二进制固件代码，难以全面分析，并高覆盖率发现 bug。

综上所述，如何针对物联网嵌入式固件实现高效自动化根因分析方法，发现固件漏洞，保障物联网嵌入固件安全，是当前亟需解决的问题。

3.2 本文研究的问题定义

3.2.1 本文关注的根本原因

现实世界中的固件崩溃原因多种多样且复杂，如使用后释放（use after free）、双重释放（double free）和越界写入（out-of-bounds write）。然而，大多数固件崩溃的原因是两个：无效的内存访问和无效的指令执行。因为它们直接关系到程

序的基本执行和内存管理机制，尤为重要，是所有计算系统都必须妥善处理的基本安全和稳定性挑战，解决这类漏洞是提高固件质量和系统安全的关键。本文选取这两个漏洞进行重点分析，将导致固件崩溃的根因定义为这两种漏洞。

无效的内存访问 无效的内存访问是指程序试图访问其没有权限访问的内存区域，主要包括以下几种：（1）越界访问：程序试图访问数组、字符串或其他数据结构的边界之外的内存；（2）野指针访问：未初始化或已释放的指针被错误地用于内存访问；（3）空指针解引用：程序试图通过一个指向 NULL 的指针访问内存或访问已经被释放的内存块。这些无效内存访问会触发操作系统的保护机制，如分页错误（page fault）或访问违规（access violation），从而导致程序崩溃或终止。同时也可能导致数据损坏或被恶意代码利用。

无效的指令执行 无效的指令执行通常指的是程序试图执行非法或非预期的机器指令。通常由以下情况几种引起：（1）代码损坏：程序代码由于某种原因（如内存损坏、磁盘故障、恶意攻击等）被更改，导致执行流程跳转到非法的指令上；（2）函数指针错误：错误的函数指针可能导致程序跳转到错误的地址执行，尤其是在使用动态链接库（DLL）或回调函数时；（3）类型错误：程序错误解析了变量类型，或应用了不匹配的类型，如将整数用作函数指针，可能导致程序尝试执行数据区域的代码。执行无效指令可能导致程序行为不可预测，从错误操作到程序崩溃都可能发生，严重时可能被用于执行恶意代码，是一种较为常见但是又较为危险的漏洞

3.2.2 内存别名问题

在分析固件崩溃的原因时，分析人员需要通过逆向分析执行轨迹，通过逆向导致崩溃过程的指令，来分析出导致崩溃的根本原因。这一过程涉及对崩溃前的数万条指令的深入分析，以追踪数据的流向和变化。在分析导致固件崩溃的执行轨迹时,需要重点考虑故障定位中的一个常见而关键的问题——内存别名问题。该问题描述的是多个指针可能指向同一块底层内存位置，在逆向时无法确定是哪个指针导致了内存的改变。这种情况增加了数据流分析的不确定性，并

且复杂化了准确追踪数据传播的任务[29][16]。

代码 3.1: 内存别名问题示例

```
; 假设edi寄存器存储的是某个对象的地址
mov eax, [edi + 4]  ; 从对象中加载第一个字段到 eax
add eax, 10         ; 对该字段增加10
mov [edi + 4], eax  ; 将修改后的值写回相同位置

; esi寄存器通过计算得到相同的内存地址
lea esi, [edi + 4]  ; 加载相同的内存地址到 esi
mov ebx, [esi]      ; 加载数据到 ebx，期望得到原始数据
```

具体地，代码 3.1是一个内存别名的简单例子，在这个例子中，$edi+4$ 和 esi 实际上指向同一内存位置，eax 通过 $edi+4$ 加载和存储数据，而 ebx 通过 esi 来加载数据。虽然 $edi+4$ 和 esi 指向同一地址，但途径不同，这会使逆向工程师或者自动化工具误为这两个操作影响的是两个独立的内存区域；也会导致我们无法分析出内存的改变是哪个指针做出的，这对代码分析影响极为严重，甚至可能导致巨大的性能下降。此外，在并发编程中，内存别名会增加数据竞态的风险。

因此，内存别名作为一项重要的问题，需要在自动化方法设计中特备关注。

3.3 物联网嵌入式固件自动化根因分析方法设计

针对嵌入式固件运行时资源有限以及调试信息有限的特点，我们提出了如 图 3.1所示的自动化根因分析框架。

本方法通过三个主要组件实现高效精准的崩溃分析：首先是高效运行时信息记录组件，在此阶段，系统通过模拟执行引起崩溃的测试用例来收集关键的运行时数据；之后是历史信息指导的逆向执行组件，此阶段利用记录的数据解决内存别名问题，确保数据流分析的准确性；最后进行根因分析组件，通过后向污点分析确定导致崩溃的具体原因，为进一步的调查提供指导。这一整体架构能够在资源受限的环境中有效地定位固件中的错误和安全漏洞。

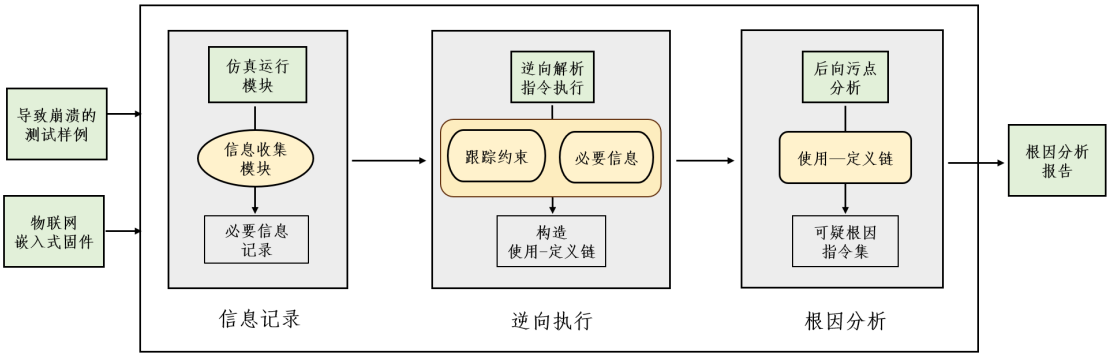

图 3.1 物联网嵌入式固件自动化根因分析方法框架

3.3.1 高效运行时信息记录

为了有效地分析固件崩溃，解决在资源受限的固件环境中调试机制不足的问题，本设计引入了一种高效的运行时信息记录的方法。本方法首先执行带有崩溃测试用例的固件二进制文件，并收集运行时数据，如寄存器状态和内存访问等，以支持随后的反向执行和根本原因分析。

为了分析固件崩溃，本方法主要识别了一种关键的事件类型：内存读取与写入。这些事件涉及到内存的读取和写入操作，如内存访问，寄存器操作等。内存读写事件在后续追踪中，可以有效辅助分析崩溃过程中数据是如何被处理和转移的。系统利用固件重置技术来执行崩溃测试用例，同时监控和记录内存读写。

代码 3.2: 内存读取与写入事件示例

```
0x403b76 STR R0, [R3, #16]        ;(a)R0:0x0, R3:0x201073c4
0x403b78 LDR R9, [R5, #16]
...
0x406c78 LDR R6, [R0, #16]        ;(b)R0:0x201073c4, R6:0x0
0x406c7a LDR R4, [R6, #8]         ;(c)Crash!
```

如代码 3.2所示，程序首先在地址 $R3+16$ 处存储 $R0$ 的值（指令 a）；然后程序加载 $R0+16$ 地址处的值并将其赋给 $R6$（指令 b），由于此处 $R0$ 值与指令

a 处 $R3$ 值相同，所以 $R6$ 的值会变为 $0x0$；接着，程序试图从 $R6+8$ 的地址加载值，由于 $R6$ 此时为 $0x0$，导致了空指针解引用的崩溃（指令 c）。

这个例子展示了一个无效读取导致的崩溃。然而，在给定崩溃测试用例和固件二进制文件的情况下，获取此类崩溃知识并不是一件简单的事情。分析人员需要深入研究运行时细节，检查从崩溃现场开始的每一条指令，最终确定 $R0+16$ 和 $R3+16$ 指向相同地址（即 $0x201073d4$），并且 $R0$ 在（指令 a）处的无效值（即 $0x0$）引发了崩溃。

为了解决这种问题对根因分析造成的困扰，我们设计了一种高效的运行时信息记录方法。首先我们在仿真环境模拟运行程序，在运行时主要关注两种类型的内存事件：（1）成功的内存读取和（2）成功的内存写入。当任何一种事件发生时，本方法会暂停仿真，捕捉相关的运行时信息，如指令地址、操作的数据、寄存器状态以及内存地址等。为了有效地保存和追踪这些事件，我们设计了一种特定的数据结构，用以记录内存访问的类型（读或写）、执行的指令地址、具体的数据内容、相应的内存地址以及寄存器的值。借助这些数据事件，我们能够获得具体的内存访问信息，而非仅仅恢复寄存器和内存数据。通过这种基于事件的方法，我们能够有效地构建出一幅详细的崩溃现场“画像”。这样的信息收集为后续的逆向执行和根因分析提供了数据基础。这种方法不仅提高了信息收集的效率，而且由于是在控制环境中进行，还增加了分析的可靠性和精确性。

3.3.2 历史信息指导的逆向执行

逆向执行阶段的主要目的是构建崩溃测试用例中数据的动态传播路径。为了解决前文提到的可能会混淆数据流分析的普遍问题——内存别名，本方法采用了一种历史驱动的逆向执行方法。这种方法通过利用在信息记录阶段捕获的具体内存访问信息，能够有效并准确地解决内存别名问题，从而使得数据依赖关系的追踪更加精确。这种逆向执行策略增强了对数据传播路径的理解，提高了根因分析的精度和效率。

对于导致崩溃的无效内存访问或无效指令执行，确定根本原因的基本思路

是从崩溃处开始逆向执行轨迹中的指令，以查明导致崩溃的数据为何被污染，这个过程被称为逆向执行。然而，即便拥有完整的执行轨迹，精确地确定数据传播和相关联个别指令也是面临着较大的困难。这主要是因为内存别名问题。例如，再考虑 代码 3.2的例子，其中指令（a）和指令（b）访问相同的内存地址（即$0x201073d4$）。如果我们能够建立这种关联，那么我们就可以将指令（a）视为崩溃的根本原因。然而，主要困难在于这些指令涉及的具体内存地址常常由于不可逆指令的存在而变得模糊不清。以指令 $MOV\ R3,\ \#0$ 为例，它将寄存器 $R3$ 设置为 $0x0$，在逆向执行时我们只知道 R3 在这条指令被执行后的值是 $0X0$，但无法恢复这条指令执行前 $R3$ 的原始值。

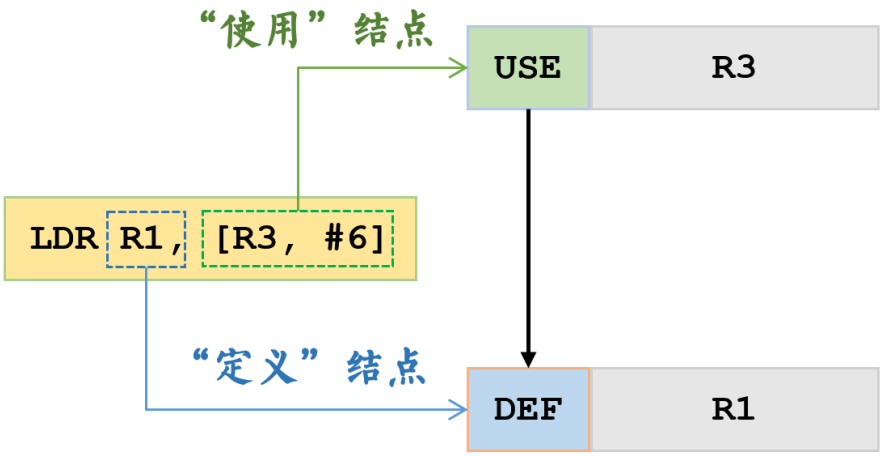

图 3.2 一个简单的使用定义链示例

为了应对这一难点，我们设计了一个历史信息指导的逆向执行方法。首先，我们构建使用-定义链来分析执行轨迹，此结构能帮助我们追踪每个操作的影响，明确数据是如何从一个操作传递到另一个操作的。然后，在使用-定义链的基础上，我们进行了逆向执行指令以恢复寄存器到之前的状态。如 图 3.2是一个简单的使用定义链示例。

（1）使用——定义链构建 这个过程从迭代重构完整执行轨迹开始，对于遇到的每条指令，我们会仔细解析指令的行为，提取源操作数和目标操作数，这些操作数代表了指令内的数据流。下面以 代码 3.3所示的程序为例展示如何构建使

用——定义链：

代码 3.3： 构建使用——定义链过程代码示例

```
MOV R1, #4        ; (a) 将数字4加载到寄存器R1
MOV R2, #10       ; (b) 将数字10加载到寄存器R2
ADD R3, R1, R2    ; (c) 将R1和R2的值相加，结果存入R3
STR R3, [R4]      ; (d) 将R3的内容存储到R4指向的内存地址
```

首先对于指令 (a)：$MOV\ R1,\ \#4$，涉及到的定义节点为 $R1$。对于指令 (b)：$MOV\ R2,\ \#10$，涉及到了对 $R2$ 的定义。对于指令 (c)：$ADD\ R3,\ R1, R2$，其中 $R1$，$R2$ 的值被读取并用于计算，因此它们是使用节点；同时这条指令定义了 $R3$ 的值，所以 $R3$ 是一个定义节点。对于指令 (d)：$STR\ R3,\ [R4]$，$R3$ 的值被用于存储操作，而 $R4$ 是作为地址的基础，因此它们都是使用节点; 同时由于这个操作定义了内存地址 $[R4]$ 的内容，所以 $R4$ 是一个定义节点。直观的，我们可以用图 3.3如所示来表述数据的流动。在这个链中，每个箭头代表数据流动的方

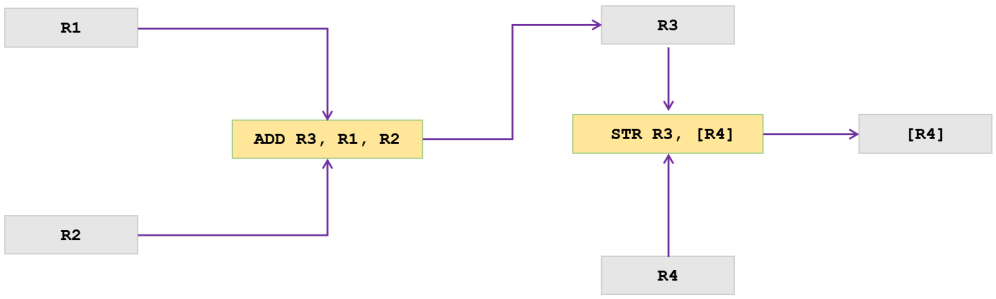

图 3.3 根据代码 3.3构建的可视化数据流动方向

向，从定义节点到使用节点，然后可能再到另一个定义节点。根据上述分析，我们可以绘制出一个使用-定义链的简化图，如代码 3.4所示。

代码 3.4： 根据代码 3.3构建的可视化使用——定义链

```
(R1) ---> (ADD R3, R1, R2) --> (R3)
(R2) ---> (ADD R3, R1, R2) --> (R3)
```

```
(R3) ---> (STR R3, [R4])
(R4) ---> (STR R3, [R4]) ---> [R4]
```

通过分析每条新指令的语义，我们可以识别所有相关的源和目标操作数。同时随着执行轨迹中每条新指令的解析，相应的使用和定义节点被添加到不断增长的使用-定义链中。通过这种方式，我们可以清晰地追踪程序中各个值的来源和去向，这对于理解程序的行为以及调试和分析程序中的错误非常有帮助。

（2）数据恢复 在构建了使用-定义链后，逆向执行过程的下一步是逆向恢复每条指令执行前的寄存器值。为了实现这一目标，我们设计了多种处理程序来处理特定类型的指令。例如，我们为加法操作设计了 $add\_handler$ 函数。以指令 $addR0, R1$ 为例，我们可以推导出以下三种关系：（1）$R0' = R0 + R1$，可以获得 $R0'$ 的值，即指令被执行后 $R0$ 的值；（2）$R0 = R0'R1$，可以逆向恢复执行前 R0 的值。（3）$R1 = R0'R0$，可以逆向恢复 R1 的值。如果在三个操作数中只有一个未知值，我们可以使用其他两个值来恢复它，并更新相应的节点。每当使用-定义链中的一个节点被更新时，本方法会检查整个链条，寻找其他可能的更新来传播。这个迭代过程持续进行，直到使用-定义链收敛，此时本方法已尽可能多地恢复了之前的状态。

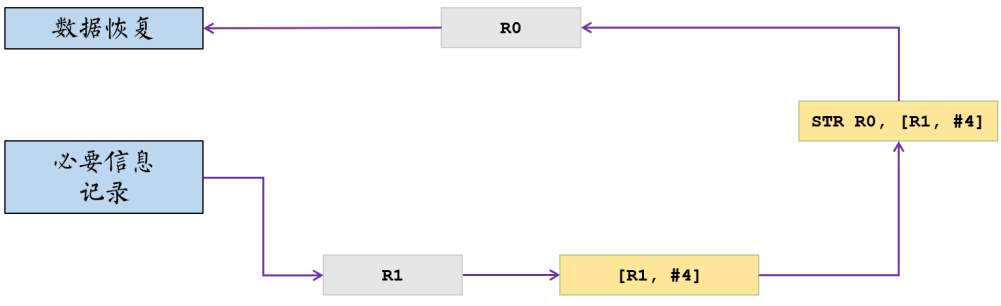

图 3.4 根据第一阶段记录信息恢复运行时数据示例

值得注意的是，这种方法无法处理内存别名问题。为了解决这个问题，我们利用第一阶段记录的信息来恢复实际的运行时数据。例如，对于指令 $STR\,R0, [R1, \#4]$，

它将 $R0$ 的值存储在地址 $R1+4$ 的内存中。

如图3.4所示,在之前的足迹收集阶段,我们已经获得了目标地址(即 $[R1, \#4]$)和要存储的具体数据。有了这个数据事件，我们可以直接从具体数据中恢复 R0 的值，并通过从目标地址减去 4 字节来恢复 R1 的值。与假设测试和值集分析等其他技术相比，这种方法不仅保证了精确性，还大大加快了逆向执行的速度。

3.3.3 根因分析

在这一节中，我们介绍本文方法的最后一个环节——根因分析。首先，我们从嵌入式固件崩溃处开始执行后向污点分析，以识别与根本原因相关的指令。这一分析通过跟踪与崩溃相关的数据的污点传播，来确定可能导致崩溃的指令，从而使分析人员可以更有效地聚焦于最关键的指令，更精确地定位和理解崩溃的原因。

本阶段主要用到的方法是后向污点分析。后向污点分析从崩溃点开始，逆向追溯数据的来源，以确定是哪些操作和值导致了程序崩溃，从而确定根因。基于此信息，我们通过逆向执行轨迹，从崩溃点开始追踪数据的反向流动。下面将详细解释后向污点分析环节。

（1）污点汇的定义和作用 污点汇（Taint Sink）是后向污点分析中定义的一个术语，它指的是分析的起始点，通常是内存地址或寄存器，它们直接关联到错误或异常的发生。在内存读写错误的情况下，涉及的寄存器或内存位置成为污点汇；如果是由特定的指令如跳转或调用导致的崩溃，涉及的寄存器（如程序计数器 PC）或相关栈地址被标记为污点汇。如果一条指令试图从一个未初始化的寄存器读取数据或写入到一个非法的内存地址，这个寄存器或内存地址就会被识别为污点汇。

代码 3.5: 污点汇定义示例

```
xor edi, edi
mov eax, 1234
mov [edi], eax
```

在后向污点分析中，我们会寻找与错误直接相关的内存地址或寄存器，这些就是所谓的污点汇。如 代码 3.5所示，在这个简单的示例中，edi 被清零，而指令 $mov\ [edi],\ eax$ 试图将 eax 寄存器中的值写入到 edi 寄存器指向的内存地址，这会导致访问违规错误。因此在这个例子，edi 寄存器是污点汇，在后向污点分析过程中，我们会从 $mov\ [edi],\ eax$ 开始，追踪导致 edi 被设为零的原因，从而揭示导致程序崩溃的根本原因。

值得注意的是，这个例子中 edi 被简单的清零来模拟非法或被保护的内存地址，但在更为复杂的程序中，某些寄存器或内存地址可能由于错误的程序逻辑或错误的内存管理而未正确初始化或被错误地修改，从而导致了程序崩溃。

（2）后向传播规则 本方法中，污点传播规则是用来追踪和分析程序执行过程中数据的流动，尤其是污点汇反向追踪到根因的过程。这些规则是本方法进行后向污点分析的核心部分，帮助定位和诊断导致程序崩溃或错误行为的具体指令或操作。

代码 3.6: 后向污点分析算法

```
def backward_taint_analysis(crash_site, instructions):
    # 1. 识别污点汇
    taint_sinks = identify_taint_sinks(crash_site)
    # 2. 初始化污点集合
    tainted_instructions = set(taint_sinks)
    # 3. 反向追踪污点
    while tainted_instructions:
        current_instruction = tainted_instructions.pop()
        # 查找当前指令的来源指令，并将其加入污点集合
        for source_instruction in find_source_instructions(
   current_instruction, instructions):
            if source_instruction not in tainted_instructions:
                tainted_instructions.add(source_instruction)
```

```
    # 4. 报告污点分析结果
    report_tainted_instructions(tainted_instructions)
def identify_taint_sinks(crash_site):
    # 根据崩溃点识别污点汇
    # 例如，根据崩溃类型识别涉及的寄存器或内存地址
    return [crash_site]
def find_source_instructions(instruction, instructions):
    # 根据指令寻找可能的源头指令，例如使用到相同数据的先前指令
    return [inst for inst in instructions if inst.destination ==
   instruction.source]
def report_tainted_instructions(tainted_instructions):
    # 输出被标记为污点的指令
    print("Tainted instructions identified:")
    for inst in tainted_instructions:
        print(inst)

# 示例用法
instructions = load_instructions()  # 加载程序中所有指令
crash_site = get_crash_site()  # 获取崩溃发生的指令或位置
backward_taint_analysis(crash_site, instructions)
```

如 代码 3.6是后向污点分析的伪代码。本方法中，我们从污点汇出发，使用构建好的使用-定义链追踪数据的来源。下面是污点传播的详细过程：首先，我们根据使用定义链，识别哪些操作使用了污点汇中的数据，将污点汇转换为使用节点；之后对于每一个使用节点，我们查找导致这些使用的定义节点，这些定义节点是之前的指令或操作，它们生成或改变了使用节点中的数据；然后从定义节点开始，我们继续追踪这些节点的使用节点，然后再找到这些使用节点的定义节点。这个过程是递归进行的，通过不断地追踪定义和使用的关系，这一过程持续

进行，直到没有新的节点可以添加到污点集合中，或者已追溯到数据流的起点。这意味着所有可能与错误相关的数据流都被标识和分析了；最后，完成污点传播后，我们集合所有的污点指令，这些指令代表了可能的错误源。通过进一步的分析，可以确定哪些指令最有可能是导致错误的根本原因。

3.4 整体系统实现

本工作伪代码如 代码 3.7所示，

代码 3.7: 整体系统工作方法

```
def backward_taint_analysis(crash_site, instructions):
    # 识别污点汇
    taint_sinks = identify_taint_sinks(crash_site, instructions)
    # 构建使用-定义链
    use_define_chain = build_use_define_chain(instructions)
    # 执行反向污点传播
    tainted_instructions = propagate_taint(taint_sinks,
   use_define_chain)
    # 输出可能的错误源
    report_potential_root_causes(tainted2_instructions)
# 主程序流程
instructions = load_instructions()  # 加载程序指令
crash_site = get_crash_site()  # 获取崩溃发生的位置
backward_taint_analysis(crash_site, instructions)
```

3.5 本章小结

4 实验与评估

5 总结与展望

参考文献

[1] Cost of a Data Breach Report 2023[J].
[2] HP-Wolf-Security-Threat-Insights-Report-Q1-2022[J].
[3] CHEN J, DIAO W, ZHAO Q, et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-Based Fuzzing[C]//Proceedings 2018 Network and Distributed System Security Symposium. Internet Society, 2018.
[4] LIU C, YAN X, FEI L. SOBER: Statistical Model-Based Bug Localization[J].
[5] LI W, SHI J, LI F, et al. μAFL: Non-Intrusive Feedback-Driven Fuzzing for Microcontroller Firmware[C]//Proceedings of the 44th International Conference on Software Engineering. 2022.
[6] WANG Q, CHANG B, JI S, et al. SyzTrust: State-Aware Fuzzing on Trusted OS Designed for IoT Devices[Z]. 2023.
[7] Embedded Trace Macrocell Architecture Specification[J]. 2011.
[8] KOSCHER K, KOHNO T, MOLNAR D. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems[J].
[9] ZALEWSKI M. American Fuzzy Lop[Z]. https://lcamtuf.coredump.cx/afl/. Apr. 2024.
[10] ABREU R, ZOETEWEIJ P. On the Accuracy of Spectrum-Based Fault Localization[J].
[11] ABREU R, ZOETEWEIJ P, VAN GEMUND A. An Evaluation of Similarity Coefficients for Software Fault Localization[C]//2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). IEEE, 2006.
[12] JONES J A, HARROLD M J. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique[C]//Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, 2005.
[13] BLAZYTKO T, SCHLÖGEL M, ASCHERMANN C, et al. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation[J].
[14] ARUMUGA NAINAR P, CHEN T, ROSIN J, et al. Statistical Debugging Using Compound Boolean Predicates[C]//Proceedings of the 2007 International Symposium on Software Testing and Analysis. ACM, 2007.
[15] WU R, ZHANG H, CHEUNG S C, et al. CrashLocator: Locating Crashing Faults Based on Crash Stacks[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014.
[16] CUI W, PEINADO M, CHA S K, et al. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps[C]//Proceedings of the 38th International Conference on Software Engineering. ACM, 2016.
[17] SHANG X, CHENG S, CHEN G, et al. How Far Have We Gone in Stripped Binary Code Understanding Using Large Language Models[Z]. 2024.
[18] YANG A Z H, LE GOUES C, MARTINS R, et al. Large Language Models for Test-Free Fault Localization[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, 2024.
[19] MAKHSHARI A, MESBAH A. IoT Bugs and Development Challenges[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.
[20] HAO D, XIE T, ZHANG L, et al. Test Input Reduction for Result Inspection to Facilitate Fault Localization[J]. Automated Software Engineering, 2010, 17.
[21] ABREU R, ZOETEWEIJ P, GOLSTEIJN R, et al. A Practical Evaluation of Spectrum-Based Fault Localization[J]. Journal of Systems and Software, 2009, 82.
[22] XU D, TANG D, CHEN Y, et al. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples[J].
[23] ARTZI S, KIM S, ERNST M D. ReCrash: Making Software Failures Reproducible by Preserving Object States[G]//VITEK J. ECOOP 2008 – Object-Oriented Programming: vol. 5142. Springer Berlin Heidelberg, 2008.

[24] CAO Y, ZHANG H, DING S. SymCrash: Selective Recording for Reproducing Crashes [C]//Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ACM, 2014.
[25] BELL J, SARDA N, KAISER G. Chronicler: Lightweight Recording to Reproduce Field Failures[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.
[26] OHMANN P. Making Your Crashes Work for You (Doctoral Symposium)[C]//Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, 2015.
[27] XU J, MU D, CHEN P, et al. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016.
[28] CAI Y, HAN B, LI J, et al. ModelCoder: A Fault Model based Automatic Root Cause Localization Framework for Microservice Systems[C]//2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). 2021.
[29] CUI W, GE X, KASIKCI B, et al. REPT: Reverse Debugging of Failures in Deployed Software[J].
[30] AHO A V, AHO A V. Compilers: Principles, Techniques, & Tools[M]. 2nd ed. Pearson/Addison Wesley, 2007.
[31] 丰田汽车（中国）投资有限公司召回部分进口雷克萨斯 RX 汽车[EB/OL]. 2023. https://www.lexus.com.cn/node/5705.
[32] KA-SAT Network cyber attack overview[EB/OL]. 2022. https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview.

作者简历

姓名：张乔　**性别：**男　**民族：**汉

出生年月：2002 年 7 月 **籍贯：**黑龙江省大庆市

2017.09-2020.07 大庆实验中学

2020.09-2024.07 浙江大学攻读学士学位

获奖情况：

参加项目：

发表的学术论文：

本科生毕业论文（设计）任务书

一、题目：

二、指导教师对毕业论文（设计）的进度安排及任务要求：

起讫日期 20　年　月　日　至　20　年　月　日

指导教师（签名）_____ 职称 _____

三、系或研究所审核意见：

负责人（签名）_____

年　月　日

本科生毕业论文（设计）考核

一、指导教师对毕业论文（设计）的评语：

指导教师（签名）_____

年　月　日

二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：

成绩比例	文献综述（10%）	开题报告（15%）	外文翻译（5%）	毕业论文质量及答辩（70%）	总评成绩
分值					

负责人（签名）_____

年　月　日

第二部分

毕业论文开题报告